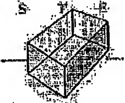
RED BRICK

SYSTEMS



WHITE

PAPER

STAR SCHEMA PROCESSING FOR COMPLEX QUERIES

RED BRICK®
The Data Warehouse Company™

INTRODUCTION

The growth of data warehousing has been led not by marketing hype but by the real value of the information that organizations everywhere are deriving from their previously untapped data.

Organizations implementing data warehouses find that the methods and tools that served them so well with operational systems often fall short of the demands of the data warehouse. This is particularly true of traditional relational database management systems (RDBMSs). Systems designed for on-line transaction processing (OLTP) do not easily address the requirements of data warehousing. In particular, the data models and schemas that have been so successful for OLTP systems are not well-suited to ad hoc analysis of complex data.

Ongoing experience in data warehousing helps identify better schemas for complex queries. The star schema, in particular, has gained widespread acceptance for the intuitive nature of its design. Unfortunately, traditional OLTP systems are not designed for the multidimensional nature of the star schema and as a result provide poor performance on complex queries to this schema. This paper describes the causes and nature of these performance problems, and the family of technologies that Red Brick has implemented to address these issues in the data warehouse environment.

STAR SCHEMAS

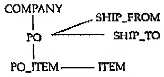
After many years of experience, businesses in all industries have become adept at implementing efficient operational systems such as payroll, inventory tracking, and purchasing. Businesses have long known how to define RDBMS schemas that allow operational systems to operate at peak efficiency, expediting a large number of small, but simultaneous read/write requests. Schema definition often focuses on defining tables that map very efficiently to operational requests while minimizing contention for access to individual records. In other words, operational systems maximize concurrency and optimize insert/update/delete performance.

The demands placed on the database by a data warehouse are very different. A data warehouse typically needs to process queries that are large, complex, ad hoc and data-intensive. Not only are there significant technological differences in how these systems consume computing resources, but the nature of what is being done requires a fundamentally different approach to defining the database schema.

These differences are best illustrated by a simple example. Figure 1 shows a typical OLTP schema for an order entry system. This schema works quite well for an operational system that can efficiently create, update, and process individual purchase orders (POs); however, it is not well suited for analyzing purchase patterns. For example, an analyst structuring a query to list company name, cost of goods, source location and destination would have to spend some time figuring out how to navigate the various tables; the process would not be intuitive.

Data warehouses require a new query-centric view of the data; star schemas provide such a view. The basic premise of star schemas is that information can be classified into two groups:

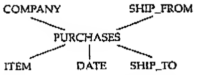
Figure 1. Typical OLTP Schema



facts and dimensions. Facts are the core data element being analyzed. In our example, purchases of individual items are facts, while in a point-of-sale (POS) database sales are facts. Dimensions are attributes about the facts. In our example, these are the item purchased and the date purchased, or in a POS database, the date of sale and the item sold. Most, if not all, analysis is based on these dimensions and hence the term dimensional analysis.

Given this two-way classification of information, Figure 2 shows an alternative schema for a purchase order database. Asking the same business question against this schema is much more straightforward because we are looking up specific facts (PURCHASES) through a set of dimensions (SHIP_FROM, SHIP_TO, ITEM). It's also significant that, in the typical star schema, the fact table is much larger than any of its dimension tables. This point becomes important as we consider the performance issues associated with star schemas later in this paper.

Figure 2. Star Schema



The intuitive structure of a star schema makes it better suited to the data warehouse environment than traditional OLTP schemas. A data warehouse must respond to a barrage of ad hoc queries. Its success depends, in part, on the queries being easy to formulate—analysts cannot spend hours navigating a maze of interrelated tables. The star schema is rapidly gaining acceptance as the schema of choice for data warehouse applications because of its intuitive approach to presenting data.

The purchasing example is relatively simple: other real world applications are more complex, requiring multiple linked fact tables which share common dimension tables. Extended through multiple fact tables, the star schema can handle the most complex environments, such as those in the health care or telecommunications industries.

POTENTIAL PERFORMANCE PROBLEMS WITH STAR SCHEMAS

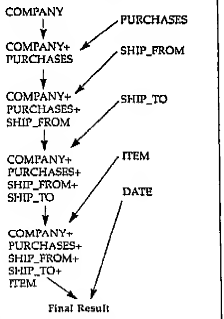
Because most experts agree that star schemas are the preferred modeling method for data warehousing, you might think that the database issues are more or less resolved. Unfortunately, this is not the case. As implementors have tried to host intuitive query-centric schemas on traditional OLTP databases, they have encountered a major obstacle: poor performance.

PAIR-WISE JOIN PROBLEM

Traditional OLTP database engines are not designed for the rich set of complex queries that can be issued against a star schema. In particular, the need to retrieve related information from several tables in a single query—"join processing"—is severely limited. Traditional OLTP databases typically join two tables at a time. If a complex join involves more than two tables, the RDBMS needs to artificially break the query into a series of pair-wise joins. Pair-wise joining is suitable for the simple requests of an OLTP system, but cannot perform adequately in a data warehouse environment.

The limitation of the pair-wise join is best illustrated in an example. Using our example star schema (Figure 2), if you wanted to list the company name, purchase price, source location, destination location, and purchase date, you would need to join data from six tables: COMPANY (for the company name), PURCHASES (for the purchase price), SHIP_FROM (for the source city), SHIP_TO (for the destination city), ITEM (for the item in question), and DATE (for the purchase date). A traditional OLTP database would have to select two tables to join initially, such as COMPANY and PURCHASES. These two tables would be joined and generate an intermediate result consisting of COMPANY joined to PURCHASES. This intermediate join result would be joined with another table, perhaps SHIP_FROM, to produce another intermediate join result. This process would continue, generating four intermediate results before creating the full result (see Figure 3). These intermediate results can be large and very costly to create.

Figure 3. Pair-Wise Joining


COMPANY
PURCHASES
COMPANY+ PURCHASES
SHIP_FROM
COMPANY+ PURCHASES+ SHIP_FROM
SHIP_TO
COMPANY+ PURCHASES+ SHIP_FROM+ SHIP_TO
ITEM
DATE
COMPANY+ PURCHASES+ SHIP_FROM+ SHIP_TO+ ITEM
Final Result

JOIN ORDER PROBLEM

The order in which the joins are done dramatically affects query performance. As an extreme example, if the join of ITEM to PURCHASES results in the selection of only a single record, subsequent joins would only be joining this one record. However, if PURCHASES is joined to COMPANY first, an intermediate result might contain every single row in PURCHASES. Because selecting the order of the pair-wise joins can have such a dramatic performance impact, traditional OLTP databases waste a great deal of effort on finding the best order in which to execute those joins.

Because the number of combinations to be evaluated grows exponentially with the number of tables being joined, the problem of selecting the best order of pair-wise joins rarely can be solved in a reasonable amount of time. The number of ways to pair-wise join a set of N tables is N![1] For example, a six-table query has 6! = (6x5x4x3x2x1) = 720 combinations. For a seven-table query, there would be (7x6x5x4x3x2x1) = 5,040 combinations; for a ten-table query there would be 3,628,800 combinations, and so on.

Even these numbers are misleading because when evaluating a join between two tables, the RDBMS may have many different join algorithms with which the tables could be joined. Each of these algorithms may need to be evaluated for every combination. For example, if there are five possible join algorithms, the database may need to evaluate 10!x5 = ~18 million combinations for a ten-table query. This problem is so serious that some databases—for example, DB2 on an IBM mainframe—will actually refuse to run a query that tries to join too

[1] Kiyoshi Ono and Guy M. Lohman, "Measuring the complexity of Join Enumeration in Query Optimization," proceedings of the 16th VLDB Conference, Brisbane, Australia; August 13-16, 1990.

many tables. With all traditional OLTP databases, it's also important to remember that the task of deciding the pair-wise join order must be completed before the query even begins to execute. Any time the database spends evaluating pair-wise join combinations is time *not spent* running the query and answering the business question.

STAR SCHEMA JOIN PROBLEM

Because the number of pair-wise join combinations is often too large to evaluate fully, traditional OLTP databases pick an "interesting" subset for evaluation. How these subsets are selected differs from one database to the next, but in general the system starts by picking combinations of tables that are directly related. Using our example star schema, joining ITEM and PURCHASES would be interesting but joining ITEM and COMPANY would not. This strategy works reasonably well with traditional OLTP schemas that contain a rich network of interrelated tables, but unfortunately fails miserably for star schemas.

Looking only at directly related tables doesn't work for data warehouses because, in a typical star schema, the only table directly related to most other tables is the fact table. This means that the fact table is a natural candidate for the first pair-wise join. Unfortunately, the fact table is typically the very largest table in the query, so this strategy invariably leads to selecting a pair-wise join order that generates a very large intermediate result set. Generating large intermediate result sets severely affects query performance. Furthermore, these large intermediate sets can fill up available space, stopping further query execution.

Clearly, these problems only compound in a complex star schema situation, where you may query multiple linked fact tables, and have even larger intermediate results.

LIMITATIONS IN OPTIMIZATION TECHNIQUES FOR COMPLEX QUERIES

Traditional cost-based optimizers don't deal well with complex queries. The basic premise of these optimizers is that you can estimate the cost of a query before executing it. Unfortunately, for complex queries in data warehouse environments, this premise does not always hold true.

Traditional OLTP databases generate cost estimates based on statistics gathered from the data itself. These statistics can only supply crude estimates for the complex queries that are typical of the data warehousing environment. Worse yet, a poor estimation early in the process can result in a much larger deviation, in effect magnifying the initial error as the query proceeds.

Take as an example the problem of data skew: data that is not uniformly distributed among its various dimensions. Cost-based optimizers typically generate cost estimates based on the size of intermediate results. These decisions can be led astray by dependencies between the data values. For example, an optimizer that knows the percentage of customers in California and the percentage of customers purchasing convertibles will operate as if the variables are independent, when in fact, California may account for a higher percentage of convertibles, relative to its size, than North Dakota or the state of Washington.

The net effect is that, even if the traditional OLTP database happens to stumble onto an optimal strategy, it may dismiss it as too costly because of errors in the cost estimation process.

PSEUDO-SOLUTIONS TO THE PERFORMANCE PROBLEMS

The performance problems described here have not gone unnoticed by the traditional OLTP database providers who are beginning to offer "solutions" to these critical problems. Unfortunately for the data warehouse implementor, such solutions are constrained by the very infrastructures that have made these RDBMSs so successful in OLTP applications. Nonetheless, it is important to understand these pseudo-solutions and their limitations.

PICKING BETTER JOIN PAIRS

A common optimization that provides some relief for the star schema join problem is evaluating more combinations of pair-wise join orderings. The basic idea is to try to get around the pair-wise join strategy of only selecting related tables.

To understand how this works, you first need to understand what it means to join unrelated tables. When two tables are joined and no columns "link" the tables, every combination of the two tables' rows are produced. In RDBMS-speak, this is called a "Cartesian product." For example, if the ITEM table had two rows ("paper," "tape") and the COMPANY table had three rows ("Sears," "KMart," "Walmart") the Cartesian product would contain six rows: "paper"+ "Sears," "tape"+ "Sears," "paper"+ "KMart," "tape"+ "KMart," "paper"+ "Walmart," and "tape"+ "Walmart." Normally, the RDBMS would never consider Cartesian products as reasonable pair-wise join candidates, but for star schemas, there are times when these Cartesian products improve query performance.

Generating Cartesian products can sometimes improve query performance because the fact table in a star schema is typically much larger than its dimension tables. Remember that one of the problems with selecting related tables to join is that the fact table is chosen very early on for a pair-wise join. This can be a very bad choice because it may generate a large intermediate result due to the sheer size of the fact table. Alternatively, if a Cartesian product of all dimension tables is first generated (by successive pair-wise joins), the join to the fact table can be deferred until the very end. The key benefit is that the large fact table does not find its way into any of the intermediate join results. The key cost is the need to generate the Cartesian product of the dimension tables. As long as the cost of generating the Cartesian product is less than the cost of generating intermediate results with the fact table, this optimization has some benefit.

This simple optimization trick clearly isn't a panacea. Remember that although the query against the star schema is naturally a multi-table join, the RDBMS is still artificially breaking it down into a set of pair-wise joins. Worse yet, this strategy is only viable if the Cartesian product of dimension rows selected is much smaller than the number of rows in the fact table. To illustrate using our example schema, if there were 10,000 ITEM rows, 500 SHIP_FROM rows, 500 SHIP_TO rows, 3,000 DATE rows, and 1,000 COMPANY rows, the final intermediate result size could be 7,500 trillion rows![2] Using this strategy, processing would probably never complete, and the RDBMS would have to use the more traditional pair-wise join ordering. The multiplicative nature of the Cartesian join makes the optimization helpful only for relatively small problems. This last point comes close to the heart of the matter: "small" and "data ware-

[2] 10,000 x 500 x 500 x 3,000 x 1,000 = ~7,500 trillion
In practice, these databases would not use this approach in this case, but would revert instead to pair-wise joins or other methods. This example does highlight the limited usefulness of this approach with large dimension tables.

house" are not synonymous.

IMPROVING STAR SCHEMA PERFORMANCE WITH PARALLELISM

Many traditional OLTP database vendors tout parallelism as the answer to performance problems for data warehousing. On the surface, parallelism appears particularly attractive in light of the ever-growing number of high-performance symmetric-multiprocessing (SMP) and massively-parallel processing (MPP) systems on the market. Unfortunately for business users, parallelism—although an important component of a viable data warehousing RDBMS—is not sufficient to overcome the limitations of an RDBMS designed for OLTP.

Through parallelism, users can either reduce the execution time of a single query (speed-up) or handle additional work without degrading execution time (scale-up). Speed-up is achieved by applying more computing resource (CPUs/disks) to the query while not increasing the amount of data the query needs to process. Scale-up is achieved by applying enough additional computing resource to the query to compensate for the additional data the query will need to process. Most databases have parallel query capability of one form or another and are able to achieve these objectives to varying degrees.

Unfortunately, these objectives completely ignore the basic cost-performance criterion central to commercial information systems. When applied to traditional brute force join strategies, parallelism tries to solve the performance problem by throwing enough dollars (computing resources) at the problem to make execution time bearable. This is best illustrated by an example:

Assume that the best pair-wise algorithm processes a certain multi-table query in 500 seconds. If we assume that this algorithm is perfectly parallelizable, a ten-processor SMP system would process the query in 50 seconds. Not bad, although there are other alternatives. For example, suppose there is a "super algorithm" that is ten times more efficient at processing multi-table joins than the pair-wise join algorithm. This super algorithm would process the same query in 50 seconds on a single processor. The parallelized super algorithm on the ten-processor system would process the query in an amazing five seconds!

Clearly, the super algorithm wins. You get the same 50-second performance time as with the ten-processor system, but you save the cost of the other nine processors or you use the processors to run other queries.

In fact this analysis paints too rosy a picture for traditional databases: traditional systems don't scale perfectly with additional processors. All current traditional OLTP databases have overhead associated with parallelism that results in less than linear scale-up. For example, going from 10 CPUs to 20 CPUs may only reduce the execution time of a 60-minute query to 45 minutes (a 25 percent speedup instead of the 50 percent you might expect). As more CPUs are added, the problem usually gets worse and, in the extreme, adding more CPUs may actually slow down the query.

Furthermore, OLTP parallel implementations do not have a strategy for applying the right degree of parallelism automatically for queries: parallelism on demand, as it were. For example, one query may benefit from a small degree of parallelism, while adding extra processors will actually waste resources without improving performance. Other queries may be able to use almost all of the processors you can throw at them. These OLTP systems leave it up to the database user to manually tune each query by setting the appropriate degree of parallelism on a query-by-query basis. Clearly this approach depends heavily on the human expertise and intervention.

Finally, parallelism really shouldn't be considered in isolation from the rest of the system's activities. At times when the system is relatively unused, you might choose to use more parallelism because you'll have less impact on other running queries and processes. At other, heavily-used times, a query that uses a high degree of parallelism can shut down performance for other users of the system. Parallelism in a production environment should analyze the larger issue of the system's resources as part of the determination of how much parallelism to use.

BIT VECTOR "STAR" JOINS

Another technique some databases use to speed join processing is the bit vector "star" join. This join technique speeds join processing by processing multiple bit vector indexes. Some of the traditional database vendors are starting to add this kind of bitmap indexing and processing to provide performance gains.

A traditional B-tree index maintains a list of row-IDs for each possible value of a column. This technology works well for many OLTP systems, but the lists of row-IDs may be long, and performing complex processing on lists of these lists requires a lot of CPU time. A bitmap index stores information much more efficiently in many situations.

But the real benefit of the bitmap index is processing speed, particularly when combining constraints. The process of performing a logical operation (AND or OR, for example) on a series of bitmap indexes is very efficient, particularly compared with performing similar processes on lists of row-IDs.

Bit vector star joins create bitmap lists of rows meeting the various restrictions in a query, and then intersect those rows for a new bitmap to all of the rows meeting all of the restrictions. This kind of processing is very efficient when compared to processing long lists of row-ID in B-tree indexes.

Bit vector joins are one interesting solution to multidimensional joins. But this approach by itself does have a number of limitations:

- Bit vector star joins work well for a limited class of queries; they do not provide a comprehensive solution for complex quéries.
- They don't address the issues of data skew.
- They don't adjust the degree of parallelism appropriate for the query.
- They don't address the estimation problems with traditional cost-based optimizers.
- Because this strategy uses bit vector indexes, it shares the restrictions of those indexes. For example, traditional bit vector indexes only work well with a small number of domain values. As a result, bit vector target joins only work well on those kinds of columns.

Although they represent one method for handling complex query processing, bit vector star joins are by no means a comprehensive solution to the complex queries that data warehousing generates.

RED BRICK'S FAMILY OF STAR SCHEMA PROCESSING TECHNOLOGIES

At this point, you may be thinking that all is lost, that RDBMSs are poor candidates for data warehouses, and that other non-RDBMS solutions must be pursued. Fortunately, the relational model is well suited to data warehousing. The standardization of SQL and the wealth of available tools and expertise reinforce the idea that a RDBMS is the natural choice for the data repository of the data warehouse.

What is needed is a new way to efficiently process complex queries against data warehouse databases: a way that doesn't suffer from the many problems previously discussed. The complex query processing required in data warehousing environments presupposes a comprehensive and flexible approach to star schema processing. Red Brick's RDBMS, by virtue of having been designed from its core to deal with data warehousing environments, is uniquely positioned to deliver such a solution. Red Brick has implemented a a combination of unique, high-performance indexes, advanced optimization techniques and sophisticated join processing algorithms to address the problem of multi-table joins.

Red Brick's singular focus on data warehousing delivers a star schema solution unsurpassed in the range and sophistication of technology it offers for multidimensional joins. These include:

- Red Brick's unique STARjoin™ processing
- TARGETjoin™ technology as a complementary star schema processing technique
- Dynamic incremental optimization™ —a radically different cost-based optimizer that selects the best strategy for each query
- TARGETindex™ —advanced bitmap index technology
- Parallelism-on-demand™ —intelligent parallelism for optimal performance

By providing multiple, high-performance alternatives for multidimensional joins and an advanced optimizer for selecting among them, Red Brick Warehouse solves a wide range of join problems with optimal performance. This range of star schema processing solutions gives Red Brick the flexibility to handle real-world performance issues—problems such as unexpected queries, widely-varying user loads, and skewed data.

It's not enough to provide only one or maybe two of the components of the Red Brick solution. A comprehensive and well-integrated solution is required to provide the flexibility and performance required in the data warehouse environment.

Let's look individually at each component of Red Brick's star schema processing solution.

RED BRICK'S STARJOIN

Red Brick's STARjoin is a high-speed, single-pass, parallelizable multi-table join. Red Brick's unique STARjoin technique overcomes the problems that plague traditional OLTP database products. Moreover, even when joining only two tables, STARjoin is unlike any join method implemented by traditional OLTP databases. These differences translate into unequaled join-processing performance.

In understanding the advantage of STARjoin, it is useful to compare it to another time-honored technique for improving query performance: indexing. The use of indexes to accelerate query processing has long been a standard capability incorporated into all RDBMS products. When indexes are defined on selected columns of a table and the query constrains on those columns, the RDBMS can use the index to very quickly identify the rows of interest. In a similar way, Red Brick's RDBMS supports the creation of specialized indexes, called STARindex™, to dramatically accelerate join performance.

Red Brick's RDBMS is the only RDBMS to support these specialized indexes for join processing. The STARindex is unlike any index structure present in any other RDBMS. In other words, you cannot gain the benefits that Red Brick provides with a STARindex by using any combination of traditional B-tree or bitmap indexes.

STARindexes are created on one or more foreign key columns of a fact table. Unlike traditional indexes that contain information to translate a column value to a list of rows with that value, a STARindex contains highly compressed information that relates the dimensions of a fact table to the rows that contain those dimensions. Because STARindexes are so space-efficient, they can be built and maintained very rapidly. The STARjoin algorithm can make use of the STARindex to efficiently identify all the rows required for a particular join.

To better understand how STARjoins can use STARindexes so effectively, it is useful to contrast STARindexes with the composite key indexes often supported by traditional OLTP databases. These composite indexes consist of two or more column values within the table being indexed. When these indexes are defined and a query references these column values, the index allows very rapid selection of target rows. Similarly, the presence of a STARindex allows Red Brick's RDBMS to rapidly identify which target rows of the fact table are of interest for a particular set of dimensions. Also, because STARindexes are created over foreign keys, no assumptions are made about the type of queries which can use the STARindex. In other words, STARindexes do not limit the kind of queries or types of joins that can be done, but accelerate queries that join related tables.

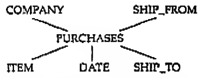
There are few similarities between STARindexes and traditional multi-column indexes. One key difference is that multi-column indexes reference a single table whereas the STARindex is the only index to reference multiple tables. Another key difference is that, with multi-column indexes, if a query does not constrain on all the columns in the composite index, the index cannot be fully used unless the specified columns are a leading subset. For example, Index(A,B,C) could be used if A+B+C are constrained, if A+B are constrained, or if just A is constrained, but not if A+C is constrained. The analogous situation for a STARindex would be a query that does not reference all the tables defined in the STARindex. Fortunately, unlike multi-column indexes, Red Brick's RDBMS can use STARindexes independent of the combination or order of constraints. Hence it lends itself to a variety of analyses resulting in different patterns of constraint processing.

Consider again the traditional OLTP database approach of generating a full Cartesian product of dimension tables, which then can be joined to the fact table. Of course, this doesn't work well for even moderately-sized dimension tables because of the size of the Cartesian product. However, what if you could get the benefits of efficiently joining the dimension tables to the fact table without the penalty of generating the full Cartesian product? The result would be the best of all possible worlds with performance far exceeding any of the traditional OLTP database approaches. The result would be a STARjoin.

STARjoin can work this seemingly impossible feat because STARindexes allow STARjoin to quickly identify which regions of the Cartesian product space contain rows of interest (in much the same way that a B-tree index can quickly identify which rows contain column values of interest). The STARjoin algorithm generates a

Cartesian product in regions where there are rows of interest and bypasses generating Cartesian products over regions where there are no rows. As a simple example, if the COMPANY dimension table contains 100 rows, a simple Cartesian product would generate rows for each of the 100 companies. However, if your fact table only contains data for two of these companies for a particular set of other dimension values, you would be unnecessarily generating 98 combinations. STARjoin would know, by virtue of STARindex, that only two companies were involved, and it would only generate Cartesian products for those two companies.

Figure 4. STARindex



A STARindex on PURCHASES could contain any combination of the foreign key fields. A single STARindex on PURCHASES that contains (COMPANY, ITEM, DATE, SHIP_FROM, SHIP_TO) would be sufficient to process any query against this star schema. However, a query that references only SHIP_FROM and SHIP_TO might be better served by a STARindex on (SHIP_FROM, SHIP_TO).

TARGETjoin

Another key component of Red Brick's STAR schema processing is TARGETjoin technology. TARGETjoin takes the simple bit vector star join technology to a whole new level of sophistication, a level that is essential for real-world data warehouse processing.

TARGETjoin relies on Red Brick's TARGETindexes, a high performance, low-maintenance bit vectored index technology already thoroughly integrated in the Red Brick RDBMS. TARGETjoin applies sophisticated multi-table join processing to solve a variety of query types. Among its benefits are:

- TARGETjoin is built on Red Brick's TARGETindexes; these have many advantages over simple bit-map indexes, including:
  - adaptive indexing technology which allows the indexes to conform optimally to any data skew
  - a family of indexes that deal with the full spectrum of data cardinalities
  - very low overhead in terms of storage space or update time on loads
  - run-time reordering and early-exit optimization for truly phenomenal performance
- TARGETjoins require no special tuning up-front except creating TARGETindexes on foreign keys in the fact table
- TARGETjoins are flexible in handling a wide variety of queries

The TARGETjoin technology assumes the existence of a TARGETindex on the relevant foreign keys in the fact table. A TARGETjoin uses the dimension tables and the TARGETindexes to create lists of rows for each restriction in the query, and then intersects those lists to create a list of the rows to retrieve from the fact table.

Let's step through this with an example. A query asks for the list of purchases over $2000 in the month of May by the sales or marketing departments. The query restricts the customer, item, and date dimensions, and references the fact table, PURCHASES.

Assuming that there are TARGETindexes on the item key, customer key and date key in the PURCHASES table, a TARGETjoin would do the following:

- From the ITEMS table, find items over $2000
- Join this with the TARGETindex on the item key in the PURCHASES table to generate a

list of rows matching this restriction

- Perform the first two steps for the other restrictions in the query
- Intersect the resulting lists; rows that occurred on all lists satisfy all requirements and should be retrieved from the sales table
- Join this list of rows with the relevant rows from the dimension tables for the complete result

The beauty of the solution is its flexibility; its only requirement is TARGETindexes on the foreign keys of the fact table. Intersecting the bitmap lists is an efficient operation. By only retrieving the relevant rows from the fact table, the TARGETjoin reduces I/O to the fact table significantly.

RED BRICK'S PARALLEL-ON-DEMAND™

Parallel processing can provide a real boost to complex query performance if the process supports parallelism and the optimizer applies it appropriately. Parallel processing may introduce a detrimental overhead on simpler queries or queries that cannot easily be broken into parallel processes. Red Brick's "parallel-on-demand" technology analyzes queries for the optimal degree of parallelism. Complex queries may have more parallel processing, while simpler queries require less. This state-of-the-art parallel processing technology is well-integrated in the Red Brick offering.

Should the need arise, the Red Brick database has the ability to exploit virtually unlimited parallelism. For example, the STARjoin is inherently parallel-izable. The work may be split into multiple, independent processes with a limited need to coordinate with each other. Each process is given its own piece of the "join space" to work on, which can be divided into an arbitrary number of pieces. The result is any number of parallel work assignments with no need to coordinate—the perfect recipe for scale-up.

Red Brick's parallel-on-demand technology is in sharp contrast to OLTP systems which cannot dynamically set degrees of parallelism. For example, consider two nearly identical queries. The first selects all purchases of toy trucks in the month of May; it joins in DATES and PRODUCTS tables to the SALES table. The second query is identical to the first except that it asks for all toy purchases in the month of May. It also joins the DATES and PRODUCTS tables to the SALES table.

The Red Brick analyzer will automatically invoke more parallelism to the second query, because it is selecting all products of the category toy, and less parallelism to the first query, which is only looking for toy trucks. Most traditional RDBMSs would require that you select beforehand the appropriate degree of parallelism for each query. In effect, this means that a user must tune the degree of parallelism on a query-by-query basis for optimal performance.

Red Brick's intelligent parallel-on-demand goes even further by looking not only at the query but at the current system resource usage. In cases where the system is largely idle, it will increase parallelism if it provides a benefit, but if the system is very busy, it will use less parallelism. This prevents a large, complex query from shutting down the system to other queries, and also enables it to get the best performance possible when system usage is low. By considering system resources in addition to the query's requirements, Red Brick's parallel-on-demand technology ensures the best performance for the overall system in addition to individual queries.

DYNAMIC INCREMENTAL QUERY OPTIMIZATION

As described before, traditional cost-based optimization doesn't work well for complex schemas. Red Brick has chosen a very different approach to this problem than traditional OLTP database vendors, an approach much better suited to the complex queries of the data warehousing environment.

Red Brick's database is the only RDBMS to support an incremental optimization model. The basic idea is to determine which sections of the query can be reasonably estimated, and then run. The actual results of this step (not estimates) are used to estimate the next step, eliminating the problem of propagating estimation errors. Better yet, a plan can change in mid-query if the actual data is radically different than statistics would have led you to expect. As an example, Red Brick's RDBMS uses this technique to select the appropriate STARindex only after it has become clear how many rows there are in the dimension tables to be joined. The result is far better index selection than could be achieved with any approach that relies on estimates.

RED BRICK'S STAR SCHEMA PROCESSING PERFORMANCE

The breadth of Red Brick's STAR schema processing technology provides unmatched join performance in a variety of situations. STARindexes provide unsurpassed performance for targeted queries. The TARGETjoin technology provides a good fallback solution for situations in which there are no STARindexes or the indexes don't match the query.

To better understand the significant performance benefit of Red Brick's STAR schema processing, consider this example. Assume that there are 500 possible ITEMS, 200 COMPANIES, 300 DATES, and one million PURCHASES in our database. Further, assume that a particular query selects 50 ITEMS, 20 COMPANIES, and 30 DATES that ultimately will select 1,000 of the PURCHASES records. (For simplicity, assume uniform distribution of the data—no data skew.)

A PAIR-WISE JOIN

In this example, the traditional pair-wise join might look something like this:

join PURCHASES + ITEM,

join (PURCHASES+ITEM) + DATE,

join (PURCHASES+ITEM+DATE) + COMPANY

The first join would generate approximately 100,000 rows, the second would generate approximately 10,000 rows, and the final join would generate 1,000 rows. So the total number of rows generated to process this query is 111,000.

A CARTESIAN JOIN

The Cartesian product approach would generate the full Cartesian product of 50 ITEMS, 20 COMPANIES, and 30 DATES = 50 x 20 x 30 = 30,000 rows. This Cartesian product could then be directly joined to the PURCHASES table to produce the final 1,000 rows at a total cost of processing 30,000 + 1,000 = 31,000 rows.

TARGETJOIN

Although the TARGETindex joins generate the same number of rows as pair-wise processing (111,000), bit vector processing is typically an order of magnitude faster than actually joining the rows themselves, because you're dealing with easily-manipulated bit strings instead of lists of row-IDs. So the TARGETjoin would be comparable to a join method that generates 11,000 rows.

STARJOIN

A well-constrained STARjoin would actually generate only slightly more combinations than exist in the selected rows of the PURCHASES table—on the order of 10 percent more for a total cost of 1,000 + (1,000 x 10%) = 1,100 rows.

Of course, for various reasons, using row counts is a very crude way of costing out the queries, but it does provide an idea of the relative costs of the approaches. In this example, the Cartesian product approach is roughly 3.5 times better than the simple pair-wise approach, but STARjoin is 28 times more efficient than even this approach, and a whopping 101 times more efficient than the simple approach!

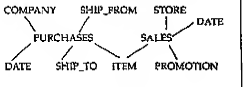
COMPLEX SCHEMAS

So far our examples have been based on a relatively simple star schema. Real-world data warehousing schemas are often more complex. Red Brick's family of technologies handles complex schemas as well as simple schemas.

As an example, consider a schema with multiple fact tables, with one or more common dimensions. Often the queries processed against schemas like this relate not only the fact tables to their associated dimensions but also the fact tables to each other. STARindexes are very well suited to this task. In particular, joining multiple fact tables can by done by selectively accessing the STARindexes of the tables being joined. The end result is a complex join being processed at speeds simply not possible with traditional OLTP databases. In fact these complex schemas are even poorer fits for the traditional OLTP databases because they have effectively doubled all the problems of the single-fact-table schema.

Figure 5 illustrates a multi-fact-table star schema.

Figure 5. Multi-Fact-Table Star Schema



The PURCHASES fact table tracks purchases of the items, and the SALES fact table tracks sales of the same items. Each fact table has dimensions which are appropriate for the analysis of those facts. The two fact tables share a common dimension, ITEM, which enables the analyst to relate the two fact tables and ask questions such as, "How do purchases of a particular set of items compare to sales of those same items?" This complex schema could present insurmountable hurdles for a traditional RDBMS, but will perform exceptionally well with Red Brick's STARjoin technology.

THE POWER OF INTEGRATION

The real power of the Red Brick family of technologies is integration and the availability of both join procedures with advanced optimization and intelligent parallelism. Well-defined STARindexes provide optimal performance for critical queries, while TARGETindexes on the foreign keys enable TARGETjoins on a wide variety of queries. Red Brick's dynamic incremental optimizer automatically selects the best algorithm for each query. Red Brick's intelligent parallelism-on-demand provides optimal performance in a parallel processing environment and uses the hardware most efficiently for each query.

SUMMARY

Data warehouses require different schemas than those used by OLTP systems; data warehouses need to provide a fast response to ad hoc, often complex queries on large amounts of data.

Data warehouses need to support incessant, intensive ad hoc queries; the star schema is very well suited to this kind of demand. OLTP databases are inherently ill-suited to the schemas and processing required by data warehousing; attempts by OLTP vendors to address their performance problems fall short of the comprehensive and flexible solution that a robust data warehouse requires.

Red Brick's RDBMS, with industry-leading breadth and depth of solutions for star schema join processing, is an ideal choice for data warehousing solutions. Its abilities to handle complex multi-fact-table schemas, assign appropriate degrees of parallelism, and dynamically choose STARindexes or other star schema processing based on the real characteristics of the data are unique in the data warehousing industry.

ABOUT RED BRICK SYSTEMS, INC.

Red Brick Systems, Inc., based in Los Gatos, Calif., is a leading provider of high-performance relational database products for data warehousing. Its flagship product, Red Brick Warehouse 5.0, is the world's fastest and most scalable relational database for data warehousing, including data marts, online analytical processing (OLAP), and data mining. Red Brick customers are successful because the company's high-performance products and service enable more users to analyze more data and make better decisions faster.

Red Brick Systems, Inc.
485 Alberto Way
Los Gatos, CA 95032
(408) 399-3200
(408) 399-3277 FAX
(800) 777-2585 (USA only)
info@redbrick.com
www.redbrick.com

P/N: 660020 6/97